IFW

Patent
45256.00033

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re the Application of: Alex Brown, et al.	**Group Art Unit:** 2183
Serial No.: 10/748,317	**Examiner:** Unknown
Filed: December 29, 2003	
For: METHOD AND APPARATUS FOR PERFORMING NATIVE BINDING	

TRANSMITTAL LETTER

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

Enclosed please find the following documents:

- Original Certified Priority Document from United Kingdom No. 0316531.3 dated 15 July 2003;
- Original Certified Priority Document from United Kingdom No. 0320717.2 dated 4 September 2003;
- Return Postcard.

No fee is believed due with this submission; however, the Commissioner is authorized to charge any fee required, or to credit any overpayment, to our Deposit Account No. **50-2613**.

Respectfully submitted,

PAUL, HASTINGS, JANOFSKY & WALKER LLP

Dated: 10/5/04

By: _____
Bradley D. Blanche, Reg No. 38,387

PAUL, HASTINGS, JANOFSKY & WALKER
Customer No. 36,183.
P.O. Box 919092
San Diego, CA 92191-9092
Telephone: (714) 668-6255
Facsimile: (714) 979-1921

CERTIFICATE OF MAILING

I hereby certify that these papers and any fees being referred to as attached or enclosed are being deposited with the United States Postal Service as first class mail. in an envelope addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on October 5, 2004.

Debbie Dean-Cross *October 5, 2004*
Date

PTO/SB/21 (09-04)
Approved for use through 07/31/2006. OMB 0651-0031
U.S. Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE

TRANSMITTAL FORM

(to be used for all correspondence after initial filing)

Total Number of Pages in This Submission

Application Number	10/748,317
Filing Date	12-29-2003
First Named Inventor	Alex Brown, et al.
Art Unit	2183
Examiner Name	
Attorney Docket Number	45256.00033

ENCLOSURES *(Check all that apply)*

- ☐ Fee Transmittal Form
  - ☐ Fee Attached
- ☐ Amendment/Reply
  - ☐ After Final
  - ☐ Affidavits/declaration(s)
- ☐ Extension of Time Request
- ☐ Express Abandonment Request
- ☐ Information Disclosure Statement
- ☒ Certified Copy of Priority Document(s)
- ☐ Response to Missing Parts/ Incomplete Application
  - ☐ Response to Missing Parts under 37 CFR 1.52 or 1.53
- ☐ Drawing(s)
- ☐ Licensing-related Papers
- ☐ Petition
- ☐ Petition to Convert to a Provisional Application
- ☐ Power of Attorney, Revocation Change of Correspondence Address
- ☐ Terminal Disclaimer
- ☐ Request for Refund
- ☐ CD, Number of CD(s) _____
  - ☐ Landscape Table on CD
- ☐ After Allowance Communication to TC
- ☐ Appeal Communication to Board of Appeals and Interferences
- ☐ Appeal Communication to TC **(Appeal Notice, Brief, Reply Brief)**
- ☐ Proprietary Information
- ☐ Status Letter
- ☒ Other Enclosure(s) (please Identify below): Return postcard

Remarks

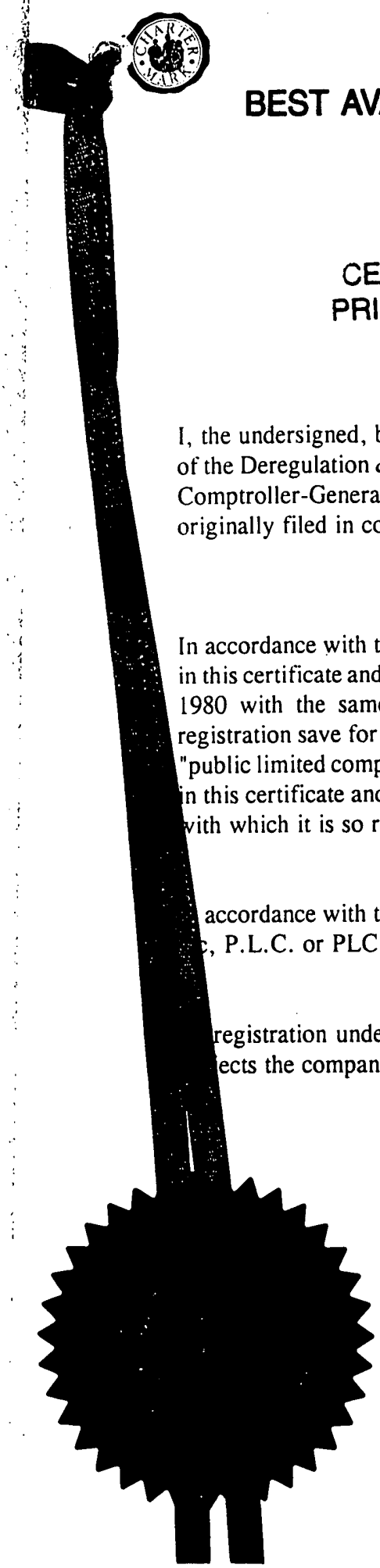
SIGNATURE OF APPLICANT, ATTORNEY, OR AGENT

Firm Name	Paul, Hastings, Janofsky & Walker LLP		
Signature			
Printed name	Bradley D. Blanche		
Date	October 5, 2004	Reg. No.	38,387

CERTIFICATE OF TRANSMISSION/MAILING

I hereby certify that this correspondence is being facsimile transmitted to the USPTO or deposited with the United States Postal Service with sufficient postage as first class mail in an envelope addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on the date shown below.

Signature			
Typed or printed name	Debbie Dean-Cross	Date	October 5, 2004



The Patent Office

INVESTOR IN PEOPLE

The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

CERTIFIED COPY OF PRIORITY DOCUMENT

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

accordance with the rules, the words "public limited company" may be replaced by p.l.c., , P.L.C. or PLC.

registration under the Companies Act does not constitute a new legal entity but merely ects the company to certain additional company law rules.

Signed

Dated 17 March 2004

Patents Form 1/77

THE PATENT OFFICE
B
15 JUL 2003
NEWPORT

The Patent Office
Cardiff Road
Newport
NP9 1RH

Request for grant of a patent

16JUL03 E822773-1 D02846
P01/7700 0.00-0316531.3

1. Your Reference	**IMR/CEE/Y1383**	
2. Application number	15 JUL 2003	**0316531.3**
3. Full name, address and postcode of the or each Applicant Country/state of incorporation (if applicable)	**Transitive Limited** **5th Floor Alder Castle** **10 Noble Street** **London** **EC2V 7QJ** **Incorporated in: United Kingdom**	
0866421 1EC1		
4. Title of the invention	**Method and Apparatus for Performing Native Binding**	
5. Name of agent	**APPLEYARD LEES**	
Address for service in the UK to which all correspondence should be sent	**15 CLARE ROAD** **HALIFAX** **HX1 2HY**	
Patents ADP number	**190001** ✓	

6. Priority claimed to:	Country	Application number	Date of filing

7. Divisional status claimed from:	Number of parent application	Date of filing
	-	-

8. Is a statement of inventorship and of right to grant a patent required in support of this application?	**YES**

9. Enter the number of sheets for any of the following items you are filing with this form. Do not count copies of the same document

Continuation sheets of this form	
Description	33
Claim(s)	2
Abstract	1
Drawing(s)	4

10. If you are also filing any of the following, state how many against each item

Priority documents	-
Translation of priority documents	-
Statement of inventorship and right to grant a patent (PF 7/77)	-
Request for a preliminary examination and search (PF 9/77)	-
Request for substantive examination (PF 10/77)	-
Any other documents (please specify)	-

11. We request the grant of a patent on the basis of this application.

Signature

APPLEYARD LEES

Date

14 July 2003

12. Contact

Ian Robinson- 01422 330110

METHOD AND APPARATUS FOR PERFORMING NATIVE BINDING

The subject invention relates generally to the field of computers and computer software and, more particularly, to program code conversion methods and apparatus useful, for example, in code translators, emulators and accelerators which translate program code that includes function calls.

In both embedded and non-embedded CPU's, one finds predominant Instruction Set Architectures (ISAs) for which large bodies of software exist that could be "accelerated" for performance, or "translated" to a myriad of capable processors that could present better cost/performance benefits, provided that they could transparently access the relevant software. One also finds dominant CPU architectures that are locked in time to their ISA, and cannot evolve in performance or market reach. Such architectures would benefit from "Synthetic CPU" co-architecture.

Program code conversion methods and apparatus facilitate such acceleration, translation and co-architecture capabilities and are addressed, for example, in the co-pending patent application, UK Application No. 03 09056 0, entitled Block Translation Optimizations for Program Code Conversion and filed on April 22, 2003, the disclosure of which is hereby incorporated by reference.

A subject program to be translated usually consists of multiple units of subject code, including the subject executable itself and a number of subject libraries, some

of which may be proprietary and some of which are provided as part of the subject OS ("system libraries"). As the subject program runs, control flow passes between these different units of subject code as function calls are made to external libraries. In some circumstances, native (i.e., target architecture) versions of certain subject libraries may be available on the target architecture.

According to the present invention there is provided an apparatus and method as set forth in the appended claims. Preferred features of the invention will be apparent from the dependent claims, and the description which follows.

The following is a summary of various aspects and advantages realizable according to various embodiments according to the invention. It is provided as an introduction to assist those skilled in the art to more rapidly assimilate the detailed design discussion that ensues and does not and is not intended in any way to limit the scope of the claims that are appended hereto.

In particular, the inventors have developed an optimization technique directed at expediting program code conversion, particularly useful in connection with a run-time translator which employs translation of subject program code into target code. A native binding technique is provided for inserting calls to native functions during translation of subject code to target code, such that function calls in the subject program to subject code functions are replaced in target code with calls to native equivalents of the same functions. Parameters of native function calls are transformed from target code

representations to be consistent with native code representations, native code calling conventions, and native function prototypes.

Native binding is the mechanism that enables translated subject code to execute a native (i.e., target architecture) version of a library directly, rather than translating and executing the equivalent subject library. This avoids the overhead of translating the subject versions of those libraries. In addition, the native version may be a much more efficient implementation of the same functionality, as the native version can exploit architectural features of the target which the subject version is unaware of.

The accompanying drawings, which are incorporated in and constitute a part of the specification, illustrate presently preferred implementations and are described as follows:

Figure 1 is a block diagram of apparatus wherein embodiments of the invention find application;

Figure 2 is a schematic diagram illustrating native binding processes in accordance with an illustrative embodiment of the invention;

Figure 3 is a schematic diagram illustrating native binding processes in accordance with an illustrative embodiment of the invention;

Figure 4 is a schematic diagram illustrating native binding processes in accordance with an illustrative embodiment of the invention;

Figure 5 is a schematic diagram illustrating native binding processes in accordance with an illustrative embodiment of the invention; and

Figure 6 is a flow diagram illustrating native function invocation in accordance with an illustrative embodiment of the invention.

Illustrative apparatus for implementing various novel features discussed below is shown in Figure 1. Figure 1 illustrates a target processor 13 including target registers 15 together with memory 18 storing a number of software components 17, 19, 20, 21 and 22. The software components include subject code 17 to be translated, an operating system 20, the translator code 19, the translated code 21, the global register store 27, and a native binding mechanism 22. Translated code 21 is also referred to as target code 21. The global register store 27 is also referred to as the subject register bank 27. The translator code 19 may function, for example, as an emulator translating subject code of one ISA into translated code of another ISA or as an accelerator for translating subject code into translated code, each of the same ISA.

The translator 19, i.e., the compiled version of the source code implementing the translator, and the translated code 21, i.e., the translation of the subject code 17 produced by the translator 19, run in conjunction

with the operating system 20 such as, for example, UNIX running on the target processor 13, typically a microprocessor or other suitable computer. It will be appreciated that the structure illustrated in Figure 3 is exemplary only and that, for example, software, methods and processes according to the invention may be implemented in code residing within or beneath an operating system. The subject code, translator code, operating system, and storage mechanisms may be any of a wide variety of types, as known to those skilled in the art.

As used herein, there is a distinction between "target code" 21, which results from the run-time translation of a subject code fragment, and "native code," which is compiled directly for the target architecture. The system target libraries provided by the target operating system are an example of native code. The translation of a system subject library is an example of target code. Native code is generated external to the translator, meaning that the translator does not have an opportunity to optimize it.

In apparatus according to Figure 1, program code conversion is preferably performed dynamically, at run-time, while the translated code 21 is running. The translator 19 runs inline with the translated program 21. The translator described above is typically deployed as an application compiled for the target architecture. The subject program is translated by the translator at run-time to execute directly on the target architecture. The translator also transforms subject operating system (OS)

calls made by the subject program so that they work correctly when passed to the target OS.

In the process of generating the translated code 21, intermediate representation ("IR") trees are generated based on the subject instruction sequence. IR trees are abstract representations of the expressions calculated and operations performed by the subject program. Later, translated code 21 is generated based on the IR trees.

The collections of IR nodes described herein are colloquially referred to as "trees". We note that, formally, such structures are in fact directed acyclic graphs (DAGs), not trees. The formal definition of a tree requires that each node have at most one parent. Because the embodiments described use common subexpression elimination during IR generation, nodes will often have multiple parents. For example, the IR of a flag-affecting instruction result may be referred to by two abstract registers, those corresponding to the destination subject register and the flag result parameter.

For example, the subject instruction "add %r1, %r2, %r3" performs the addition of the contents of subject registers %r2 and %r3 and stores the result in subject register %r1. Thus, this instruction corresponds to the abstract expression "%r1 = %r2 + %r3". This example contains a definition of the abstract register %r1 with an add expression containing two subexpressions representing the instruction operands %r2 and %r3. In the context of a subject program 17, these subexpressions may correspond to other, prior subject instructions, or they may represent

details of the current instruction such as immediate constant values.

When the "add" instruction is parsed, a new "+" IR node is generated, corresponding to the abstract mathematical operator for addition. The "+" IR node stores references to other IR nodes that represent the operands (represented in the IR as subexpression trees, often held in subject registers). The "+" node is itself referenced by the subject register whose value it defines (the abstract register for %r1, the instruction's destination register). For example, the center-right portion of Figure 20 shows the IR tree corresponding to the X86 instruction "add %ecx, %edx".

As those skilled in the art may appreciate, in one embodiment the translator 19 is implemented using an object-oriented programming language such as C++. For example, an IR node is implemented as a C++ object, and references to other nodes are implemented as C++ references to the C++ objects corresponding to those other nodes. An IR tree is therefore implemented as a collection of IR node objects, containing various references to each other.

Further, in the embodiment under discussion, IR generation uses a set of abstract registers. These abstract registers correspond to specific features of the subject architecture. For example, there is a unique abstract register for each physical register on the subject architecture ("subject register"). Similarly, there is a unique abstract register for each condition code flag present on the subject architecture. Abstract

registers serve as placeholders for IR trees during IR generation. For example, the value of subject register %r2 at a given point in the subject instruction sequence is represented by a particular IR expression tree, which is associated with the abstract register for subject register %r2. In one embodiment, an abstract register is implemented as a C++ object, which is associated with a particular IR tree via a C++ reference to the root node object of that tree.

The implementation of abstract registers is divided between components in both the translator code 19 and the translated code 21. Within the translator 19, an "abstract register" is a placeholder used in the course of IR generation, such that the abstract register is associated with the IR tree that calculates the value of the subject register to which the particular abstract register corresponds. As such, abstract registers in the translator may be implemented as a C++ object which contains a reference to an IR node object (i.e., an IR tree). The aggregate of all IR trees referred to by the abstract register set is referred to as the working IR forest ("forest" because it contains multiple abstract register roots, each of which refers to an IR tree). The working IR forest represents a snapshot of the abstract operations of the subject program at a particular point in the subject code.

In basic block mode, state is passed from one basic block to the next using a memory region which is accessible to all translated code sequences, namely, a global register store 27. The global register store 27 is a repository for abstract registers, each of which

corresponds to and emulates the value of a particular subject register or other subject architectural feature. During the execution of translated code 21, abstract registers are held in target registers so that they may participate in instructions. During the execution of translator code 21, abstract register values are stored in the global register store 27 or target registers 15.

Within the translated code 21, an "abstract register" is a specific location within the global register store, to and from which subject register values are synchronized with the actual target registers. Alternatively, when a value has been loaded from the global register store, an abstract register in the translated code 21 could be understood to be a target register 15, which temporarily holds a subject register value during the execution of the translated code 21, prior to being saved back to the register store.

Thus, a subject program running under the translator 19 has two different types of code that execute in an interleaved manner: the translator code 19 and the translated code 21. The translator code 19 is generated by a compiler, prior to run-time, based on the high-level source code implementation of the translator 19. The translated code 21 is generated by the translator code 19, throughout run-time, based on the subject code 17 of the program being translated.

The representation of the subject processor state is likewise divided between the translator 19 and translated code 21 components. The translator 19 stores subject processor state in a variety of explicit programming

language devices such as variables and/or objects; the compiler used to compile the translator determines how the state and operations are implemented in target code. The translated code 21, by comparison, stores subject processor state implicitly in target registers and memory locations, which are manipulated directly by the target instructions of the translated code 21.

For example, the low-level representation of the global register store 27 is simply a region of allocated memory. This is how the translated code 21 sees and interacts with the abstract registers, by saving and restoring between the defined memory region and various target registers. In the source code of the translator 19, however, the global register store 27 is a data array or an object which can be accessed and manipulated at a higher level. With respect to the translated code 21, there simply is no high-level representation.

Figure 2 shows the different compilation units of a translated program. The translator 105 runs as an executable compiled for the native architecture, meaning both the native OS 103 and the native processor 101. The subject program 106 consists of a subject executable 107 and possibly a number of subject libraries, which may include proprietary libraries 109 and system libraries 111. The compilation units of the subject program 106 are translated into target code and executed within the translator 105.

Figure 3 shows the units of code of a translated program with native binding. The subject program 106 consists of a subject executable 107 and possibly a number

of subject libraries, which may include proprietary libraries 109 and system libraries 111. The translator uses native binding to replace subject program calls to subject system library functions 111 with calls to native system library functions 117.

For example, for a MIPS-x86 translation, the x86 system target library "libc" may define an advanced memcpy() (memory copy) routine that takes advantage of SSE2 vector operations to perform extremely fast byte copies. Under native binding, all calls to memcpy in the MIPS subject code are bound to the native memcpy(). This eliminates the cost of translating the subject (MIPS) version of the memcpy() function. In addition, the native (x86) version of memcpy() function has a much greater awareness of the intricacies of the native hardware, so it will know the most efficient way to achieve the function's desired effect.

Native binding works by detecting when the subject program's flow of control enters a subject library for which a native version exists. Rather than translating the subject library, the translator executes equivalent native code.

Alternatively, native binding may be used for more arbitrary code substitution. For example, native binding may be used to substitute a natively compiled version of a non-library function. In addition, native binding may be used to implement subject system calls on a native architecture, by replacing all calls to subject system functions with substitute functions that either implement the same functionality or act as call stubs around target

system calls. Native binding may also be applied at arbitrary subject code locations, beyond function call sites; this mechanism allows arbitrary code sequences (either target code or native code) and/or function calls to be inserted or substituted at any well-defined point in the subject program.

Bind Point Descriptions

Native binding requires the translator to correlate particular subject code functions with their native code counterparts, so that the translator knows which subject functions to bind and which native functions to bind them to. The translator can acquire this function mapping information in different ways depending on the implementation of native binding.

In one embodiment, the subject function to be bound is identified using a special purpose "bind point" description language. A bind point description includes: (a) the subject function to be bound; and (b) the corresponding native function to be bound. The translator reads bind point descriptions at the beginning of execution to identify bind points (locations to invoke native functions). During decoding of the subject program, when the translator encounters these bind points it inserts in the target code a call stub to the appropriate native function. In one embodiment, particular bind point descriptions are embedded in the translator. In another embodiment, bind point descriptions are stored in separate files which the translator reads at run-time; this allows end-users to

control the native binding mechanism by adding particular subject-to-native function mappings.

In an alternative embodiment, the native binding description language allows arbitrary bind points to be specified, meaning that a native function call can be inserted at other points in the subject code beyond subject function calls. In this alternative embodiment, a bind point description includes: (a) a defined location within the subject program (i.e., not just function call sites); and (b) the corresponding native function to be bound. For example, arbitrary bind points may be identified as: (1) the start of a function; (2) the start of a subject module's initialization code; (3) a fixed offset from a particular symbol (e.g., a fixed offset from the start of a function); (4) a fixed offset from the first text segment in a module; or (5) all calls to a particular subject function (either within a particular module, or in all modules excluding one particular module). The difference between bind point types (1) and (5) is that (1) binds the entry point of a subject function while (5) binds the function's call site.

In some embodiments, the native binding description language allows the end-user to specify relative bind points, meaning that a native function call can be inserted before, after, or instead of a bind point (e.g., a system subject function call). For example, a native binding description could specify that the native function "foo()" be invoked immediately after all calls to the subject function "bar()".

In some embodiments, code other than native function calls may be inserted at bind points. In these embodiments, a bind point description includes: (a) a defined location within the subject program; and (b) a reference to target code block or a native code function to be invoked. If the code inserted is target code, then the translator does not need to perform much of the work associated with parameter transformation and native calling conventions (described below) at the bind point; adherence to the translator-specific target code calling conventions is sufficient. Arbitrary target and native code insertion allows the translator to perform other tasks on translated programs, such as debugging and performance profiling.

In an alternative embodiment, subject-to-native function mappings are encoded in the symbol table of the subject program before run-time, in a process referred to as runtime symbol patching. Runtime symbol patching consists of replacing entries in the subject program's symbol table with special native binding markers; this requires manipulation of the subject program after it is compiled (compile-time), but before it is translated (run-time). When the translator encounters these symbol table markers at run-time, it interprets them as bind point descriptions and interprets them to identify which native function to call. In this embodiment, the identity of the subject function to be bound is implicit in the location of the marker within the symbol table: the marker is placed in the symbol table entry corresponding to a particular subject function.

In contrast to the explicit identification of bind points by bind point descriptions, in an alternative embodiment bind points are identified implicitly by a translator specific subject instruction set extension which are planted in the subject code when it is compiled (see "S-calls" below).

Parameter Transformation

When invoking a native function, the translator must conform to the calling conventions of the target architecture; by comparison, target code does not necessarily need to adhere to the target calling conventions as long as the translator adheres to some consistent calling convention throughout the target code. In addition, the translator may need to perform data transformation between the subject machine state (as represented in target code) and the native machine state (as represented in native code), both for the native function's input parameters and its return value if any. Such data transformations may include: (i) endian conversion (i.e., byte-swapping); (ii) data structure alignment; (iii) conversion between subject addresses and target addresses; and (iv) value transformation (e.g., constant conversion or value scaling).

For example, on the MIPS architecture, function parameters are passed in registers, while on the x86 architecture, parameters are passed on the stack. For a MIPS-x86 translator to invoke a native function, the x86 calling conventions requires that function parameters be moved from the subject registers to the stack.

Figure 6 illustrates the steps performed by the translator to invoke a native function. In order to invoke a native function, the translator must perform several steps: parameter setup 501; input parameter transformation 503; native function invocation 505; and result transformation 507. Parameter setup 501 refers to the target code which calculates the values of the function call parameters. Input parameter transformation 503 organizes the function call parameter values, from their target code representations, to the format and location that the native function code expects. Native function invocation 505 is the actual function call to the native function, and includes the organization of the (reformatted) parameters into the order required by the function prototype, in a manner complying with the native calling conventions. A function prototype indicates the order and type of the function's parameters, and the type of the function's return value. For example, if the native calling conventions require that arguments be passed on the stack, then target code which invokes a native function must place the arguments on the stack in the correct order and advance the stack pointer accordingly. Result transformation 507 transforms the function's return value if any; the function returns a value in a format consistent with the native architecture, which the translator converts into the representation used by the target code.

Parameter setup 501 is not exclusive to native binding: the target code must calculate the parameter values regardless of whether the function is invoked as translated subject code or as native code. In cases where the translator does not know which particular subject

registers a native function call will use (as parameter values), the translator must rectify the values of subject registers used, to ensure that the subject register bank is in a consistent state. Translator optimizations such as lazy evaluation may postpone the calculation of subject register values until those values are needed; rectification refers to the forced calculation of registers whose evaluation has been deferred. When rectified, subject register values are then stored to the subject register bank 27.

In addition to calculating the values of a function's explicit parameters (which in some cases requires rectification of all subject registers), the parameter setup code must also ensure that the subject memory space is in a consistent state, as native calls may have side effects in the form of memory accesses. In one embodiment, the IR that encodes a native function call (whether to a native call stub or to an underlying native function) rectifies the subject memory state, such that all loads and stores that would occur prior to the function call in the subject program are planted in the target code prior to the native function call, and likewise no memory accesses that should occur after the function call are planted before the native call.

"Parameter transformation" 509 is used to refer to steps 503, 505, and 507 collectively, meaning all of the respective conversion between the different data formats and calling conventions of the target code and native code. The code that performs parameter transformation 509 is referred to as a "call stub;" it consists of a minimal wrapper around the underlying native function call, whose

sole purpose is to allow the target code caller to interact with the native code callee. A single "call stub" may therefore be divided into target code and native code components. Whether parameter transformation takes place entirely in target code or partially in native code depends on the implementation of native binding.

Native Code Stubs

In some embodiments, parameter transformation for native binding is performed in part by native code. As discussed above, native code stubs have the disadvantage that they cannot be optimized by the translator. In these embodiments, target code performs some parameter transformation and invokes a native call stub, using the native calling conventions; the native code of the call stub then performs additional parameter transformation and calls the underlying native function.

Figure 4 shows a translator which uses native binding based on native code call stubs. The subject program 106 consists of a subject executable 107 and possibly a number of subject libraries, which may include proprietary libraries 109 and system libraries 111. The translator replaces calls to subject system library functions 111 with calls to native code call stubs 113. The target code which calls the native code call stubs performs parameter transformation; the native code call stubs 113 perform additional parameter transformation and parameter mapping. The native code call stubs 113 then call native system library functions 117.

Native Code Stubs: Uniform Interface

In one embodiment, native code parameter transformation is facilitated by defining a uniform call stub function interface. A uniform interface defines a fixed function signature for all native call stubs and corresponding data types, which allows the translator to invoke the call stub without any knowledge of the function signature (prototype) of the underlying native function. This allows call stubs to be implemented in a high-level programming language such as C or C++, which makes the native binding mechanism more accessible to end users of the translator.

In this embodiment, the call stub function is compiled as a native code executable which is linked to the translator executable. During execution, the translator invokes the call stub through the uniform interface, using the native calling conventions. Because the call stub interface is uniform, the target code sequence which invokes the call stub is the same for all native calls.

For example, in one embodiment, the uniform call stub interface is a C function which takes exactly two parameters, the subject address of the call site and a reference to a uniform data structure which contains all subject register values, and returns one value, the subject address of the next subject instruction that the translator should execute. The uniform data structure which is passed to the call stub always contains the current values of all subject registers; this data structure is referred to as a subject context.

In a native binding mechanism based on the uniform call stub interface, native binding is divided into several components: (i) a special IR node type which rectifies all subject register values; (ii) target code which marshals all subject registers into a uniform context structure and invokes the call stub according to native calling conventions; and (iii) the native call stub which marshals specific subject register values into function parameters and invokes the native function.

During translation, a native call site is translated into a native call IR node. A native call IR node contains dependency references to the IRs of all subject registers. These IR dependencies of the native call IR node guarantee that, in the target code generation phase, the target code corresponding to the subject register values will be generated before the native call. Translator optimizations such as lazy evaluation may postpone the calculation of subject register values until those values are needed; the native call IR dependencies inform the code generation phase that a native call stub "needs" all subject register values. As such, the translator generates target code to rectify all subject register values prior generating target code to invoke a native call stub. Likewise, the native call IR node is treated as a memory reference for purposes of code generation, such that all loads and stores which precede the function call in the subject code are (generated and) executed prior to the function call in the target code. Similarly, all loads and stores which occur after the function call in the subject code are postponed until after the native call.

If necessary, the translator includes a special abstract register to hold the native call IR. In the IR generation phase of translation, abstract registers serve as placeholders for (i.e., root nodes of) IR trees. IR trees must be linked to an abstract register or else they are not emitted as target code. In other translators, the native call IR node can be attached to an existing abstract register, such as an abstract register for the successor address (of the current block).

The target code to invoke a native call stub rectifies the subject register values and then records them in a subject context structure. Because the call stub is implemented by native code in this embodiment, the subject context must be constructed in a representation consistent with the native architecture. As such, the target code performs parameter transformation as necessary to convert the subject register values from a target code representation to a native code representation. The process of converting multiple values into a data structure representation consistent with another architecture is sometimes referred to as marshalling.

The target code constructs a subject context containing native representations of all subject register values. The target code then invokes the native call stub, passing it the subject context as a parameter. The call stub invokes the underlying native function, extracting the particular subject registers needed from the subject context and passing them to the native function as appropriate parameters. The call stub thus encodes the native function's prototype and defines the

mapping of particular subject registers to the corresponding native function parameters.

In some cases, the native function interface may be substantively different than its subject code equivalent, such that additional calculations (beyond the transformations for data representation and calling conventions) must be performed on the subject data to make it suitable for use as a native function parameter. In such cases, the call stub may perform additional parameter transformation on the subject register values. For example, the native function may expect a particular parameter in different units than its subject code equivalent. In this case, the call stub would perform a constant conversion on the appropriate subject register value prior to invoking the native function, to account for the difference in unit type for that parameter.

In embodiments which use a uniform native call stub interface, the target code indiscriminately transforms all subject registers, from the target code representation to a native code representation. The native call stub then extracts the particular (transformed) subject registers required as parameters by the native function's prototype. The native call stub may also perform additional parameter transformations to reflect differences between the subject version and native version of the function being called. In this embodiment, therefore, target code adjusts for representation differences between the target code and native code, while the call stub accounts for the signature of the particular underlying native function.

In this embodiment, a native binding description identifies a subject function to bind and the corresponding native call stub function, while the identity of the underlying native function to bind to is implicit (i.e., hard-coded) in the call stub implementation.

Native Code Stubs: Compiled Scripts

In another embodiment, native code parameter transformation is implemented using a special purpose native binding programming language ("the scripting language"). Before or during execution, the translator parses the scripting language implementation of a call stub and compiles it into a native executable module. The call stub module is then linked in with the translator executable, and the (native) call stub functions are invoked using the native calling conventions, as described above.

Native binding scripts are compiled or interpreted into an executable representation of a call stub. In one embodiment, the bind point descriptions are interpreted by a separate tool, prior to execution of the translator, into executable native code. In another embodiment, the translator itself interprets or compiles the bind point descriptions at run-time, into either executable native code or into translator IR (later generated as target code). In some embodiments, the native binding programming language is a special-purpose language which is specific to the translator.

In one embodiment, the native binding programming language includes primitives (programming language building blocks) to describe a wide range of possible parameter transformation operations, including: (i) descriptions of data types; (ii) conversion between target code and native representations of those data types; (iii) identification of native functions; (iv) mapping particular subject registers to particular function parameters; (v) memory accesses to the subject program's memory space; and (vi) basic mathematical operations. In an alternative embodiment, the scripting language includes the above primitives plus (vi) basic logical operations and (vii) global storage of temporary values across multiple native binding scripts. The implementation of these primitives (i.e. the native code generated from the scripts, by the translator or by a special tool) must be consistent with the representation and calling conventions of the target architecture.

In this embodiment, a native binding description identifies a subject function to bind and the corresponding native call stub function, while the identity of the underlying native function to bind to is hard coded in the scripting language implementation of the call stub.

Target Code Stubs

In some embodiments, parameter transformation for native binding is performed entirely in target code. In these embodiments, the translator detects native binding calls at decode-time and encodes the parameter transformations as IR trees (which are ultimately

generated as target code). By representing the parameter transformations and details of the native function prototype in IR, the call stub code becomes integrated into the subject code (IR of a call stub is indistinguishable from IR of subject code). This allows the translator to apply optimizations (e.g., group blocks) to the parameter transformation code. In contrast, parameter transformations performed in native code, such as the mapping of subject registers to function parameters performed in native code call stubs (described above), are external to the translator and cannot be optimized.

Figure 5 shows a translator which uses native binding based on target code call stubs. The subject program 106 consists of a subject executable 107 and possibly a number of subject libraries, which may include proprietary libraries 109 and system libraries 111. The translator replaces calls to subject system library functions 111 with target code call stubs 115. The target code call stubs 115 perform parameter transformation and parameter mapping, and then call native system library functions 117.

In these embodiments, target code invokes the bound native function directly; all parameter transformations and parameter mappings are performed by target code. In contrast to native code call stubs, in these embodiments, target code accounts for both representation transformations and the signature of the particular native function (i.e., the mapping of particular subject registers to the corresponding native function parameters). In order for target code to perform parameter mapping, the translator must know the native

function prototype and subject-register-to-parameter mappings during translation.

In some cases, the target code call stub is translated in a separate block from the target code which invokes it ("caller target code"). In other cases, the target code call stub is translated in the same block as the target code which invokes it, which allows the call stub code to be integrated with and optimized with the caller target code; this mechanism is referred to as "early binding". In cases where the subject program calls a particular subject function from many call sites, it is disadvantageous to inline the target code call stub at every call site because excessive memory is consumed by the resulting duplication of call stub code. In such cases, the translator maintains the target code call stub as a separate block which each translated call site invokes, rather than inlining the call stub at every call site. One optimization of native binding is to use early binding (i.e., inline call stubs) only in blocks which are very frequently executed.

Target Code Stubs: Schizo Calls

In one embodiment, target code parameter transformation is facilitated by extending the subject instruction set to include translator-specific native binding instructions (called S-Call Commands)which are inserted into the subject code when the subject code unit is compiled. When the translator decodes the subject code, it detects and interprets these instructions, and plants the appropriate IR (or target code) to perform

parameter transformation. This mechanism is referred to as Schizo calls or "S-calls".

The S-call mechanism requires support from the developers of the subject program. When the subject program is compiled, the S-call commands are encoded in the compiled subject program as subject instructions. The call stubs describing the call to the native function are not constrained to only contain S-Call commands they may include regular, legal subject instructions to aid the parameter transformation. In combination the S-call commands and the regular subject instructions encode all information and operations required for parameter transformation.

In one embodiment S-call commands are encoded in variable length instructions constructed from multiple sub-components, such that one S-Call command instruction has the size of multiple regular subject instructions. In one embodiment a S-call command begins with a sub-component which is known to be interpreted as an illegal instruction on the subject architecture, referred to as Schizo Escape. In one embodiment, S-call commands are divided into five classes: (1) marker, (2) parameter, (3) call, (4) copy, and (5) nullify. The class of the S-call command is encoded in unused bits of the leading Schizo Escape. The Schizo Escape is followed by a combination of class-specific options and/or arguments, each of which has a predefined opcode and format, and each of which is encoded as one or more words (i.e., four-byte units) in the S-call Command instruction The S-call command instruction ends with a repetition of the initial Schizo Escape.

S-call marker commands are optional markers, used in one embodiment to allow ABI specific optimizations. They indicate the subject code range (start and end) containing translator specific code (e.g. the call stub to a native function), which may or may not coincide with the full extent of a subject function. S-call markers are of two types: start and end. An S-call marker has one string argument, assigning a name to the marked point (e.g. the name of the subject function being bound).

S-call parameter (2) commands identify a value to be used as a native function parameter, and encode the appropriate parameter transformation for that value. In one embodiment, each parameter command defines the "next" parameter for the function call, by pushing the corresponding value onto the stack; in this embodiment, parameter commands must therefore be in order corresponding to the function prototype.

S-call call (3) commands encode the actual function call to the native function. Call command arguments include the location to store the function's return value, and either the name, the absolute address or the location of the address in the subject machine state of the function being called.

S-call copy (4) commands encode the operation of copying a value to or from a subject register or subject memory location and performing transformation between target code and native representation.

S-call nullify (5) commands invalidate the instructions they follow. They are used in conjunction with unconditional branches as described below, to allow proper execution of the subject program when running natively and still allow the translator to identify and interpret translator specific pieces of code.

During translation, the translator detects the Schizo Escape and decodes the S-call commands into IR representations of the corresponding parameter transformation and native function call operations. The translator integrates the parameter transformation IR into the IR forest of the current block, which is subsequently generated as target code. Encoding the parameter transformation operations as IR allows the parameter transformation code to be integrated into and optimized with the IR encoding the subject code.

S-call commands are encoded in the subject program using leading instruction opcodes (Schizo Escape) which only the translator understands; subject processors interpret them as illegal subject instructions. As such, the execution of S-call commands must be avoided when the subject program is running natively (i.e., on the subject architecture). The subject software developers can use multiple methods to allow S-call-enhanced subject programs to run natively, including (a) conditional execution and (b) branch lookahead parsing. Conditional execution (a) consists of conditional subject code which checks whether the subject program is running natively or as a translated program, based on the run-time environment, and which skips the S-call commands if running natively. Branch lookahead parsing (b) consists of unconditional branch

instructions which are planted in the subject code such as to skip all translator specific instructions (such as, but not limited to, S-call commands). When running natively, the unconditional branches are executed thereby skipping the translator specific code. When running as a translated program, the translator disregards any unconditional branch instruction which is followed by an Nullify S-call command (i.e., the unconditional branch is part of the subject instruction pattern used by the translator to identify translator specific code in decoding).

Target Code Stubs: External Schizo Stubs

In another embodiment call stubs comprised of Schizo commands and ordinary subject instructions are in separately compiled units of subject code. The special purpose native binding description language (as described above) is used to specify subject code locations as bind points. When the translator reaches such a bind point, the flow of control is diverted to execute the external schizo stub instead. From this point onwards the behavior is identical to the behavior outlined for Schizo Calls. It allows the use of S-Call commands when it is not possible to insert S-Call commands into the subject code directly (e.g. when the source code for the subject library/application is not available).

Target Code Stubs: Interpreted Scripts

In an alternative embodiment, target code parameter transformation is facilitated by a special purpose native binding implementation language ("the scripting

language"), as described above. In this embodiment, at run-time the translator interprets the native binding script into an IR representation of parameter transformation. The translator integrates the parameter transformation IR into the IR forest of the current block, which is subsequently optimized and generated as target code. Such translators must contain a front-end component that can parse and decode the scripting language.

In this embodiment, a native binding description identifies the subject function to bind and the corresponding native call stub function, while the identity of the underlying native function to bind to is hard coded in the implementation of the call stub.

Target Code Stubs: Uniform Interface

In an alternative embodiment, target code parameter transformation is facilitated by defining a uniform call stub function interface, as described above. A uniform interface defines a fixed function signature for all native call stubs and corresponding data types, which allows the translator to invoke the call stub as a native function without any knowledge of the function signature (prototype) of the underlying native function. This allows call stubs to be implemented in a high-level programming language such as C or C++, which makes the native binding mechanism more accessible to end users of the translator.

In contrast to the native code uniform call stub interface described above, in some embodiments the translator parses the call stub implementation at run-time

and interprets it into an IR representation of parameter transformation. In other words, the translator compiles call stub implementations into translator IR. The translator integrates the parameter transformation IR into the IR forest of the current block, which is subsequently optimized and generated as target code. Such translators must contain a front-end component that can parse and decode the high-level programming language, similar to a compiler.

In this embodiment, a native binding description identifies the subject function to bind and the corresponding native call stub function, while the identity of the underlying native function to bind to is hard coded in the implementation of the call stub.

Although a few preferred embodiments have been shown and described, it will be appreciated by those skilled in the art that various changes and modifications might be made without departing from the scope of the invention, as defined in the appended claims.

Attention is directed to all papers and documents which are filed concurrently with or previous to this specification in connection with this application and which are open to public inspection with this specification, and the contents of all such papers and documents are incorporated herein by reference.

All of the features disclosed in this specification (including any accompanying claims, abstract and drawings), and/or all of the steps of any method or process so disclosed, may be combined in any combination,

except combinations where at least some of such features and/or steps are mutually exclusive.

Each feature disclosed in this specification (including any accompanying claims, abstract and drawings) may be replaced by alternative features serving the same, equivalent or similar purpose, unless expressly stated otherwise. Thus, unless expressly stated otherwise, each feature disclosed is one example only of a generic series of equivalent or similar features.

The invention is not restricted to the details of the foregoing embodiment(s). The invention extends to any novel one, or any novel combination, of the features disclosed in this specification (including any accompanying claims, abstract and drawings), or to any novel one, or any novel combination, of the steps of any method or process so disclosed.

Claims

1. A method of inserting native function calls during the translation of program code, comprising:

identifying a subject function invoked by the subject program code;

identifying a native function which corresponds to the functionality of the subject function;

executing the native function instead of the subject function in the translation of the subject program code.

2. The method of claim 1, wherein the native function executing step comprises:

transforming a function parameter from a target code representation to a native code representation;

invoking the native function with the transformed function parameter according to the prototype of the native function.

3. The method of claim 2, wherein the function parameter transforming step is generating intermediate representation of the transformation.

4. The method of claim 2 or 3, wherein the function parameter transforming step is generating target code.

5. The method of claim 2, 3 or 4, wherein the native function executing step further comprises:

transforming in target code all subject register values from the target code representation to the native code representation;

invoking from target code a native code call stub function with the transformed subject registers according to a uniform call stub interface;

invoking from the native code call stub function the native function with particular subject registers and/or parameter stack according to the prototype of the native function.

6. The method of any of claims 2 to 5, wherein the function parameter transforming step and the native function invoking step are described in subject code by translator specific instructions added to the subject instruction set.

ABSTRACT

METHOD AND APPARATUS FOR PERFORMING NATIVE BINDING

A native binding technique is provided for inserting calls to native functions during translation of subject code to target code, such that function calls in the subject program to subject code functions are replaced in target code with calls to native equivalents of the same functions. Parameters of native function calls are transformed from target code representations to be consistent with native code representations, native code calling conventions, and native function prototypes.

[Figure 6]

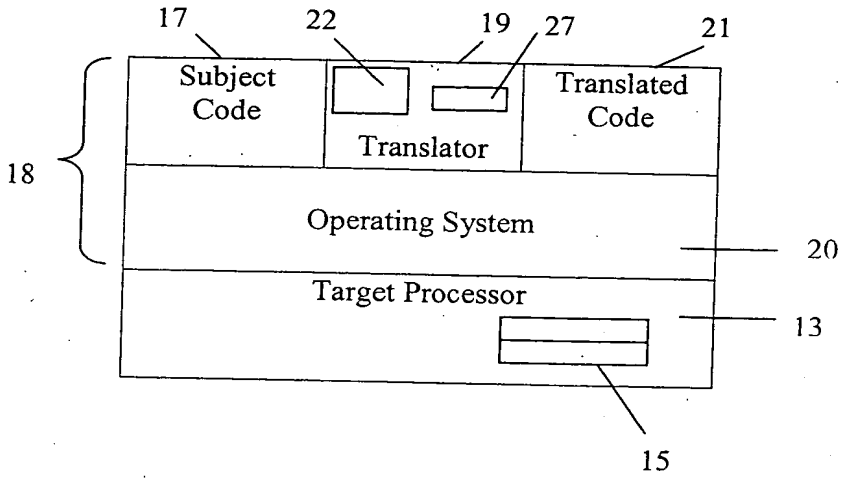

17
22
19
27
21
Subject Code
Translator
Translated Code
18
Operating System
20
Target Processor
13
15

FIG. 1

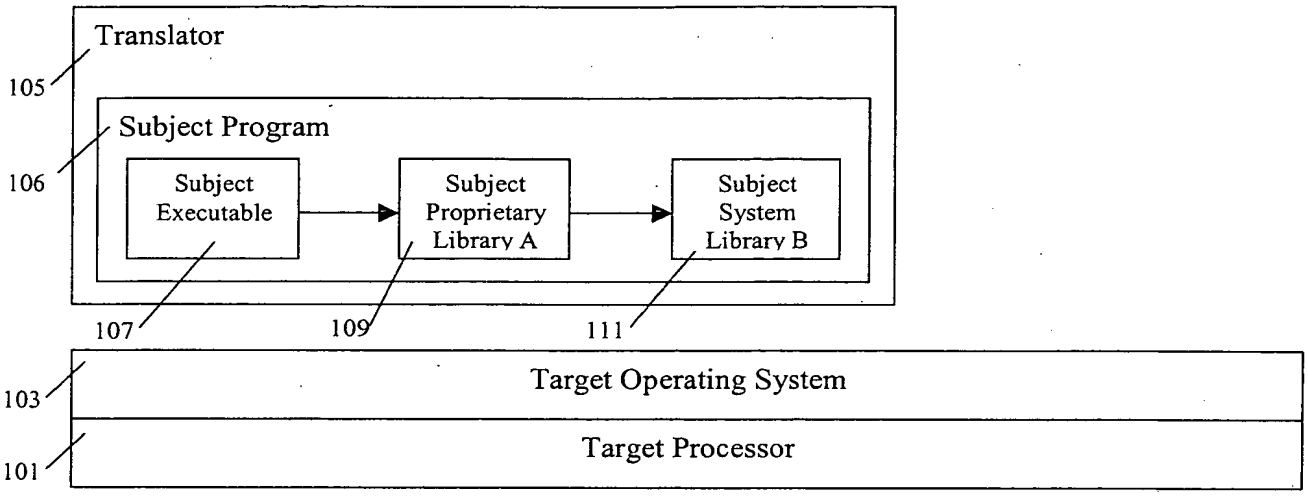

Translator
105
Subject Program
106
Subject Executable
Subject Proprietary Library A
Subject System Library B
107
109
111
Target Operating System
103
Target Processor
101

FIG. 2

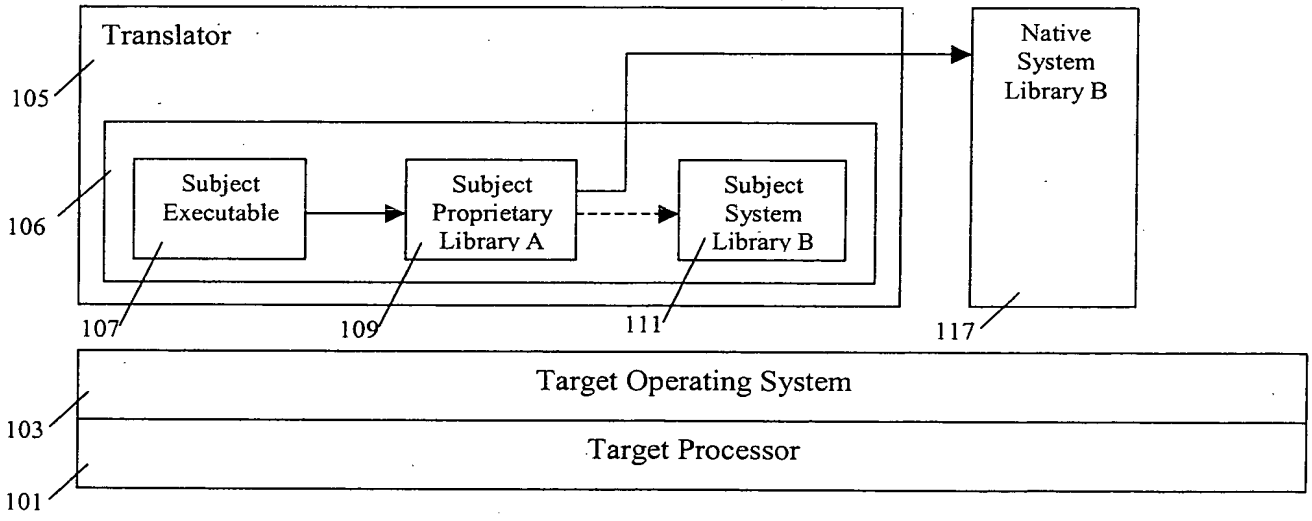

Translator
105
106
Subject Executable
Subject Proprietary Library A
Subject System Library B
Native System Library B
107
109
111
117
Target Operating System
103
Target Processor
101

FIG. 3

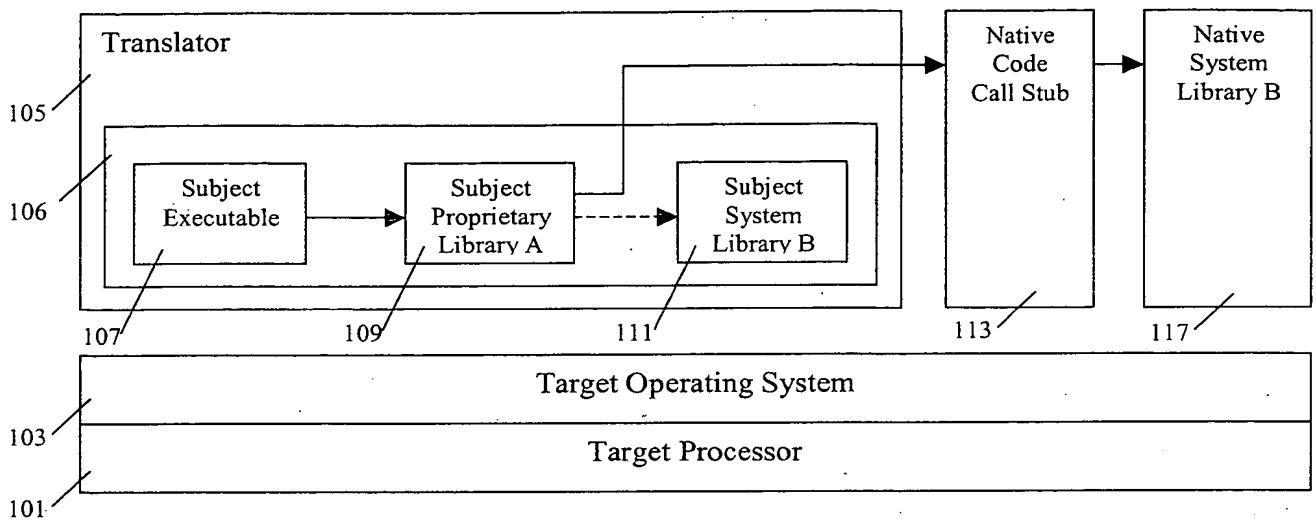

Translator
105
106
Subject Executable
Subject Proprietary Library A
Subject System Library B
107
109
111
Native Code Call Stub
113
Native System Library B
117
Target Operating System
103
Target Processor
101

FIG. 4

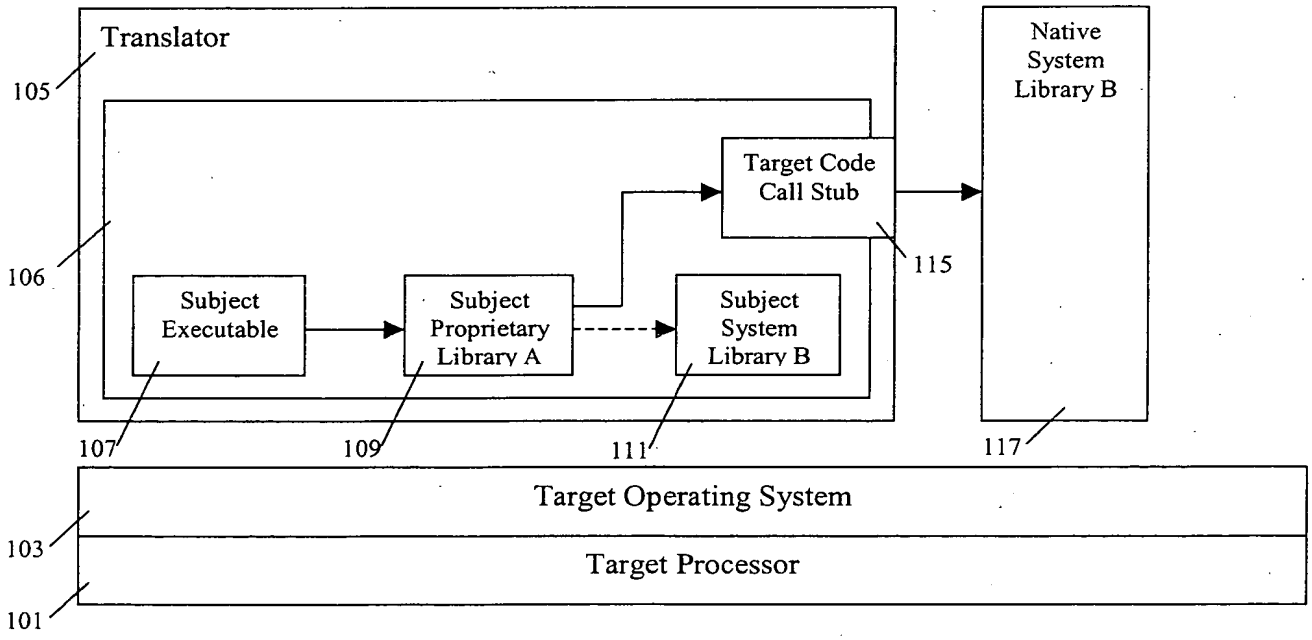

Translator
105
106
Subject Executable
Subject Proprietary Library A
Subject System Library B
107
109
111
Target Code Call Stub
115
Native System Library B
117
Target Operating System
103
Target Processor
101

FIG. 5

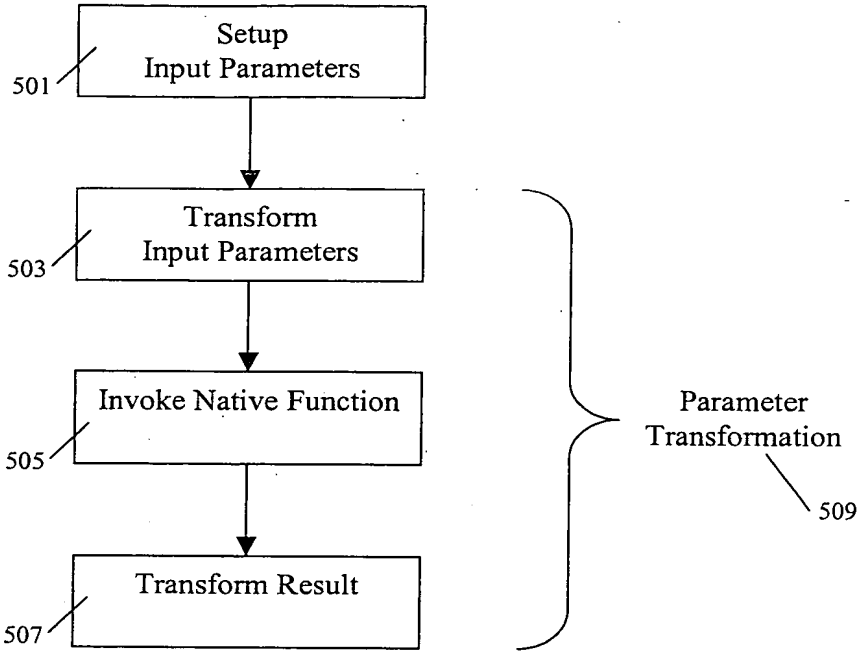

Setup
Input Parameters
501
Transform
Input Parameters
503
Invoke Native Function
505
Transform Result
507
Parameter
Transformation
509

FIG. 6